

BOOK 14: EXPRESS AND BACKEND

Chapter 1: What is Express

EXPRESS.JS

Express is a web framework for Node.js.

It simplifies building web servers and APIs.

Without Express, you write everything manually. Routing, parsing, middleware, all from scratch.

With Express, these are handled for you.

Express is:

- Minimal - Does not impose structure
- Flexible - Customize everything
- Fast - Thin layer over Node.js
- Popular - Most used Node.js framework

WHY USE EXPRESS

Compare raw Node.js to Express.

Raw Node.js:

```
const http = require("http")
const url = require("url")
```

```
let server = http.createServer((req, res) => {
  let parsed = url.parse(req.url, true)
```

```
  if (req.method === "GET" && parsed.pathname === "/") {
    res.writeHead(200, { "Content-Type": "text/html" })
    res.end("<h1>Home</h1>")
  }
})
```

```

    } else if (req.method === "GET" && parsed.pathname === "/"
users") {
    res.writeHead(200, { "Content-Type": "application/json" })
    res.end(JSON.stringify([]))
    } else if (req.method === "POST" && parsed.pathname === "/"
users") {
    let body = ""
    req.on("data", chunk => body += chunk)
    req.on("end", () => {
    res.writeHead(201, { "Content-Type": "application/json" })
    res.end(JSON.stringify({ id: 1, ...JSON.parse(body) }))
    })
    } else {
    res.writeHead(404)
    res.end("Not Found")
    }
})

```

```
server.listen(3000)
```

Express:

```

const express = require("express")
let app = express()

```

```
app.use(express.json())
```

```

app.get("/", (req, res) => {
    res.send("<h1> Home </h1>")
})

```

```

app.get("/users", (req, res) => {
    res.json([])
})

```

```

app.post("/users", (req, res) => {
    res.status(201).json({ id: 1, ...req.body })
})

```

```
app.listen(3000)
```

Express is cleaner, shorter, and easier to understand.

WHAT EXPRESS PROVIDES

Routing:

Define handlers for specific URLs and methods.

```
app.get("/users/:id", handler)
```

```
app.post("/users", handler)
```

Middleware:

Functions that run before route handlers.

```
app.use(express.json())
```

```
app.use(authenticate)
```

Request handling:

Parsed body, query, params, cookies.

```
req.body
```

```
req.query
```

```
req.params
```

```
req.cookies
```

Response helpers:

Simple methods to send responses.

```
res.send()
```

```
res.json()
```

```
res.redirect()
```

```
res.status()
```

Error handling:

Centralized error handling.

```
app.use((err, req, res, next) => {
```

```
res.status(500).json({ error: err.message })
})
```

EXPRESS IS MINIMAL

Express does not force structure.

No mandatory folder layout.

No required file names.

No built-in ORM.

No built-in authentication.

You add what you need. You structure how you want.

This flexibility is both strength and weakness. Flexibility means decisions.

EXPRESS ECOSYSTEM

Express has a rich ecosystem of middleware:

body-parser - Parse request bodies (now built in)

cookie-parser - Parse cookies

cors - Enable CORS

helmet - Security headers

morgan - Request logging

compression - Gzip compression

express-session - Session management

passport - Authentication

multer - File uploads

express-validator - Input validation

Most features are one npm install away.

HOW EXPRESS WORKS

Express is built on Node.js http module.

When a request comes in:

1. Express receives the request
2. Runs matching middleware in order
3. Finds matching route handler
4. Route handler sends response

If no route matches, Express returns 404.

Everything is a middleware in Express. Even route handlers.

ALTERNATIVES TO EXPRESS

Fastify - Faster, schema-based validation

Koa - By Express creators, async/await focused

Hapi - Configuration-driven, enterprise focused

NestJS - TypeScript, Angular-like structure

Express remains most popular due to:

- Simplicity
- Large ecosystem
- Extensive documentation
- Community size

WHEN TO USE EXPRESS

Use Express for:

- REST APIs
- Web applications
- Server-side rendering
- Microservices
- Prototyping

Express handles most web server needs.

EXPRESS PHILOSOPHY

Minimalism:

Do one thing well. Leave other decisions to you.

Middleware pattern:

Small focused functions that compose together.

Unopinionated:

No right way to structure your app.

Compatibility:

Works with any database, template engine, or frontend.

A COMPLETE EXAMPLE

```
const express = require("express")
```

```
let app = express()
```

```
app.use(express.json())
```

```
let users = []
```

```
app.get("/", (req, res) => {  
  res.send("Welcome to the API")  
})
```

```
app.get("/users", (req, res) => {  
  res.json(users)  
})
```

```
app.get("/users/:id", (req, res) => {  
  let user = users.find(u => u.id === req.params.id)
```

```
  if (user) {  
    res.json(user)  
  } else {  
    res.status(404).json({ error: "User not found" })  
  }  
})
```

```
app.post("/users", (req, res) => {  
  let user = {  
    id: Date.now().toString(),  
    ...req.body  
  }
```

```
  users.push(user)
```

```
  res.status(201).json(user)  
})
```

```
app.delete("/users/:id", (req, res) => {  
  let index = users.findIndex(u => u.id === req.params.id)
```

```
  if (index >= 0) {  
    users.splice(index, 1)  
    res.json({ success: true })  
  } else {  
    res.status(404).json({ error: "User not found" })  
  }  
})
```

```
app.listen(3000, () => {  
  console.log("Server running on http://localhost:3000")  
})
```

This is a complete REST API in about 50 lines.

PRACTICE

Understand what Express provides over raw Node.js.

Know the middleware pattern.

Understand Express is minimal and flexible.

Know the Express ecosystem.

WHAT YOU LEARNED

You learned Express is a web framework for Node.js.

You learned it simplifies routing and middleware.

You learned Express is minimal and flexible.

You learned about the middleware pattern.

You learned Express ecosystem basics.

The next chapter teaches you how to set up Express.

BOOK 14: EXPRESS AND BACKEND

Chapter 2: Setting Up Express

CREATE PROJECT

Create a new directory:

```
mkdir my-api
```

cd my-api

Initialize package.json:

npm init -y

Install Express:

npm install express

Create app.js:

```
const express = require("express")
let app = express()
```

```
app.get("/", (req, res) => {
  res.send("Hello, Express!")
})
```

```
app.listen(3000, () => {
  console.log("Server running on http://localhost:3000")
})
```

Run it:

node app.js

Visit <http://localhost:3000> in your browser.

PROJECT STRUCTURE

Basic structure:

```
my-api/
node_modules/
package.json
package-lock.json
```

app.js

As projects grow:

```
my-api/  
  node_modules/  
  src/  
  routes/  
  middleware/  
  controllers/  
  models/  
  utils/  
  app.js  
  config/  
  tests/  
  package.json
```

PACKAGE.JSON SCRIPTS

Add scripts:

```
{  
  "name": "my-api",  
  "version": "1.0.0",  
  "main": "app.js",  
  "scripts": {  
    "start": "node app.js",  
    "dev": "node --watch app.js"  
  },  
  "dependencies": {  
    "express": "^4.18.2"  
  }  
}
```

Run:

npm start

`npm run dev`

dev uses Node.js watch mode (Node 18+).

USING NODEMON

For automatic restarts:

`npm install --save-dev nodemon`

Update package.json:

```
{  
  "scripts": {  
    "start": "node app.js",  
    "dev": "nodemon app.js"  
  }  
}
```

nodemon restarts when files change.

ES MODULES

Use ES modules instead of CommonJS.

Add to package.json:

```
{  
  "type": "module"  
}
```

Change app.js:

`import express from "express"`

```
let app = express()

app.get("/", (req, res) => {
  res.send("Hello, Express!")
})

app.listen(3000, () => {
  console.log("Server running")
})
```

ENVIRONMENT VARIABLES

Install dotenv:

```
npm install dotenv
```

Create .env:

```
PORT=3000
NODE_ENV=development
```

Load in app.js:

```
require("dotenv").config()

let port = process.env.PORT ?? 3000

app.listen(port, () => {
  console.log(`Server running on port ${port}`)
})
```

BASIC APP STRUCTURE

```
const express = require("express")
require("dotenv").config()

let app = express()

app.use(express.json())
app.use(express.urlencoded({ extended: true }))

app.get("/", (req, res) => {
  res.json({ message: "API is running" })
})

app.get("/health", (req, res) => {
  res.json({ status: "ok", timestamp: new Date() })
})

app.use((req, res) => {
  res.status(404).json({ error: "Not found" })
})

app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500).json({ error: "Internal server error" })
})

let port = process.env.PORT ?? 3000

app.listen(port, () => {
  console.log(`Server running on http://localhost:${port}`)
})
```

EXPRESS APPLICATION OBJECT

app is the Express application:

```
let app = express()
```

Methods on app:

`app.get(path, handler)` - Handle GET requests
`app.post(path, handler)` - Handle POST requests
`app.put(path, handler)` - Handle PUT requests
`app.patch(path, handler)` - Handle PATCH requests
`app.delete(path, handler)` - Handle DELETE requests
`app.all(path, handler)` - Handle all methods
`app.use(middleware)` - Use middleware
`app.listen(port, callback)` - Start server

Properties:

`app.locals` - Application-wide variables
`app.mountpath` - Path where app is mounted

EXPRESS SETTINGS

Configure Express:

```
app.set("view engine", "ejs")
app.set("views", "./views")
app.set("json spaces", 2)
```

Get settings:

```
app.get("view engine")
```

Common settings:

`env` - Environment (development, production)
`view engine` - Template engine
`views` - Views directory
`trust proxy` - Trust X-Forwarded-* headers

DISABLE SETTINGS

Disable features:

```
app.disable("x-powered-by")
```

Removes X-Powered-By header.

```
app.disable("etag")
```

Disables ETag generation.

MULTIPLE LISTEN OPTIONS

Listen on port:

```
app.listen(3000)
```

With callback:

```
app.listen(3000, () => {  
  console.log("Server started")  
})
```

With host:

```
app.listen(3000, "127.0.0.1", () => {  
  console.log("Listening on 127.0.0.1:3000")  
})
```

Listen on all interfaces:

```
app.listen(3000, "0.0.0.0")
```

GRACEFUL SHUTDOWN

Handle shutdown properly:

```
let server = app.listen(3000)
```

```
process.on("SIGTERM", () => {  
  console.log("SIGTERM received, shutting down")
```

```
  
  server.close(() => {  
    console.log("Server closed")  
    process.exit(0)  
  })  
})
```

```
process.on("SIGINT", () => {  
  console.log("SIGINT received, shutting down")
```

```
  
  server.close(() => {  
    console.log("Server closed")  
    process.exit(0)  
  })  
})
```

COMPLETE STARTER TEMPLATE

```
const express = require("express")  
require("dotenv").config()
```

```
let app = express()
```

```
app.disable("x-powered-by")
```

```
app.use(express.json())  
app.use(express.urlencoded({ extended: true })))
```

```
app.use((req, res, next) => {  
  console.log(`${new Date().toISOString()} ${req.method} ${req.url}`)  
})  
next()  
})
```

```
app.get("/", (req, res) => {  
  res.json({  
    name: "My API",  
    version: "1.0.0",  
    status: "running"  
  })  
})
```

```
app.use((req, res) => {  
  res.status(404).json({ error: "Not found" })  
})
```

```
app.use((err, req, res, next) => {  
  console.error(err.stack)  
  res.status(500).json({ error: "Internal server error" })  
})
```

```
let port = process.env.PORT ?? 3000
```

```
let server = app.listen(port, () => {  
  console.log(`Server running on http://localhost:${port}`)  
})
```

```
process.on("SIGTERM", () => {  
  server.close(() => process.exit(0))  
})
```

PRACTICE

Create an Express project from scratch.

Configure package.json scripts.

Use environment variables.

Set up basic middleware.

Handle 404 and errors.

WHAT YOU LEARNED

You learned to create an Express project.

You learned package.json scripts.

You learned to use nodemon.

You learned environment variables.

You learned Express app configuration.

You learned graceful shutdown.

The next chapter teaches you about routes and methods.

BOOK 14: EXPRESS AND BACKEND

Chapter 3: Routes and Methods

BASIC ROUTING

Routes define how your app responds to requests.

```
app.method(path, handler)
```

```
app.get("/", (req, res) => {  
  res.send("GET /")  
})
```

```
app.post("/users", (req, res) => {  
  res.send("POST /users")  
})
```

```
})
```

```
app.put("/users/1", (req, res) => {  
  res.send("PUT /users/1")  
})
```

```
app.delete("/users/1", (req, res) => {  
  res.send("DELETE /users/1")  
})
```

HTTP METHODS

app.get() - Read data

app.post() - Create data

app.put() - Replace data

app.patch() - Update data

app.delete() - Remove data

app.head() - Like GET but no body

app.options() - CORS preflight

app.all() handles all methods:

```
app.all("/api/*", (req, res, next) => {  
  console.log("API request:", req.method, req.url)  
  next()  
})
```

ROUTE PATHS

String paths:

app.get("/", handler)

app.get("/about", handler)

```
app.get("/users", handler)
```

With parameters:

```
app.get("/users/:id", handler)
```

```
app.get("/posts/:postId/comments/:commentId", handler)
```

Pattern matching:

```
app.get("/ab?cd", handler)
```

```
app.get("/ab+cd", handler)
```

```
app.get("/ab*cd", handler)
```

Regular expressions:

```
app.get(/.*fly$/, handler)
```

ROUTE PARAMETERS

Access URL parameters:

```
app.get("/users/:id", (req, res) => {  
  let userId = req.params.id  
  res.send(`User ID: ${userId}`)  
})
```

Request: GET /users/123

Output: User ID: 123

Multiple parameters:

```
app.get("/users/:userId/posts/:postId", (req, res) => {  
  let { userId, postId } = req.params  
  res.json({ userId, postId })  
})
```

Request: GET /users/123/posts/456

Output: {"userId":"123","postId":"456"}

QUERY STRINGS

Access query parameters:

```
app.get("/search", (req, res) => {  
  let query = req.query.q  
  let page = req.query.page ?? "1"  
  let limit = req.query.limit ?? "10"  
  
  res.json({ query, page, limit })  
})
```

Request: GET /search?q=hello&page=2&limit=20

Output: {"query":"hello","page":"2","limit":"20"}

Query parameters are always strings.

ROUTE HANDLERS

Single handler:

```
app.get("/", (req, res) => {  
  res.send("Hello")  
})
```

Multiple handlers:

```
app.get("/", handler1, handler2, handler3)
```

Array of handlers:

```
let handlers = [authenticate, authorize, controller]
```

```
app.get("/admin", handlers)
```

Handlers must call next() to continue:

```
function handler1(req, res, next) {  
  console.log("Handler 1")  
  next()  
}
```

```
function handler2(req, res) {  
  res.send("Handler 2")  
}
```

```
app.get("/", handler1, handler2)
```

CHAINED ROUTE HANDLERS

app.route() chains handlers for same path:

```
app.route("/users")  
  .get((req, res) => {  
    res.json([])  
  })  
  .post((req, res) => {  
    res.status(201).json(req.body)  
  })
```

```
app.route("/users/:id")  
  .get((req, res) => {  
    res.json({ id: req.params.id })  
  })  
  .put((req, res) => {  
    res.json({ id: req.params.id, ...req.body })  
  })  
  .delete((req, res) => {  
    res.json({ deleted: true })  
  })
```

```
})
```

EXPRESS ROUTER

Create modular route handlers:

```
const express = require("express")
let router = express.Router()
```

```
router.get("/", (req, res) => {
  res.json([])
})
```

```
router.get("/:id", (req, res) => {
  res.json({ id: req.params.id })
})
```

```
router.post("/", (req, res) => {
  res.status(201).json(req.body)
})
```

```
module.exports = router
```

Mount in main app:

```
const userRoutes = require("./routes/users")
```

```
app.use("/users", userRoutes)
```

Now:

```
GET /users -> router.get("/")
```

```
GET /users/123 -> router.get("/:id")
```

```
POST /users -> router.post("/")
```

ORGANIZING ROUTES

routes/users.js:

```
const express = require("express")
let router = express.Router()
```

```
let users = []
```

```
router.get("/", (req, res) => {
  res.json(users)
})
```

```
router.get("/:id", (req, res) => {
  let user = users.find(u => u.id === req.params.id)
  user ? res.json(user) : res.status(404).json({ error: "Not found" })
})
```

```
router.post("/", (req, res) => {
  let user = { id: Date.now().toString(), ...req.body }
  users.push(user)
  res.status(201).json(user)
})
```

```
router.put("/:id", (req, res) => {
  let index = users.findIndex(u => u.id === req.params.id)
```

```
  if (index >= 0) {
    users[index] = { id: req.params.id, ...req.body }
    res.json(users[index])
  } else {
    res.status(404).json({ error: "Not found" })
  }
})
```

```
router.delete("/:id", (req, res) => {
  let index = users.findIndex(u => u.id === req.params.id)
```

```
  if (index >= 0) {
    users.splice(index, 1)
```

```
res.json({ deleted: true })
} else {
res.status(404).json({ error: "Not found" })
}
})
```

module.exports = router

routes/index.js:

```
const express = require("express")
let router = express.Router()

router.use("/users", require("./users"))
router.use("/posts", require("./posts"))
router.use("/comments", require("./comments"))
```

module.exports = router

app.js:

```
const routes = require("./routes")

app.use("/api", routes)
```

ROUTER MIDDLEWARE

Add middleware to router:

```
router.use((req, res, next) => {
  console.log("Router middleware")
  next()
})
```

Or specific to route:

```
router.get("/:id", validateId, getUser)
```

PARAMETER MIDDLEWARE

Run for specific parameters:

```
router.param("id", (req, res, next, id) => {
  let user = users.find(u => u.id === id)

  if (!user) {
    return res.status(404).json({ error: "User not found" })
  }

  req.user = user
  next()
})

router.get("/:id", (req, res) => {
  res.json(req.user)
})

router.put("/:id", (req, res) => {
  Object.assign(req.user, req.body)
  res.json(req.user)
})

router.delete("/:id", (req, res) => {
  let index = users.indexOf(req.user)
  users.splice(index, 1)
  res.json({ deleted: true })
})
```

ROUTE ORDERING

Routes match in order defined:

```
app.get("/users/new", handler1)
app.get("/users/:id", handler2)
```

/users/new matches handler1
/users/123 matches handler2

If reversed, /users/new would match /:id first.

Put specific routes before parameterized routes.

PRACTICE

Create routes for different methods.

Use route parameters and query strings.

Organize routes with `express.Router`.

Use route chaining.

Handle route ordering.

WHAT YOU LEARNED

You learned HTTP method routing.

You learned route paths and parameters.

You learned query string access.

You learned multiple route handlers.

You learned `express.Router` for organization.

You learned route ordering importance.

The next chapter teaches you about middleware.

BOOK 14: EXPRESS AND BACKEND

Chapter 4: Middleware

WHAT IS MIDDLEWARE

Middleware is a function that runs between request and response.

```
function middleware(req, res, next) {  
  next()  
}
```

It has access to:

- req - Request object
- res - Response object
- next - Call next middleware

Middleware can:

- Execute code
- Modify request and response
- End the request-response cycle
- Call next middleware

MIDDLEWARE FLOW

Request -> Middleware 1 -> Middleware 2 -> Route Handler -> Response

Each middleware calls next() to continue.

If middleware does not call next(), request stops there.

```
app.use((req, res, next) => {  
  console.log("Middleware 1")  
  next()  
})
```

```
})
```

```
app.use((req, res, next) => {  
  console.log("Middleware 2")  
  next()  
})
```

```
app.get("/", (req, res) => {  
  console.log("Route handler")  
  res.send("Done")  
})
```

Output:

Middleware 1

Middleware 2

Route handler

APPLICATION-LEVEL MIDDLEWARE

Runs for all requests:

```
app.use((req, res, next) => {  
  console.log(`${req.method} ${req.url}`)  
  next()  
})
```

Runs for specific path:

```
app.use("/api", (req, res, next) => {  
  console.log("API request")  
  next()  
})
```

Runs for specific method and path:

```
app.get("/users", authenticate, (req, res) => {  
  res.json([])
```

```
})
```

BUILT-IN MIDDLEWARE

`express.json()` - Parse JSON bodies:

```
app.use(express.json())
```

```
app.post("/users", (req, res) => {  
  console.log(req.body)  
  res.json(req.body)  
})
```

`express.urlencoded()` - Parse URL-encoded bodies:

```
app.use(express.urlencoded({ extended: true }))
```

`express.static()` - Serve static files:

```
app.use(express.static("public"))
```

LOGGING MIDDLEWARE

Simple logger:

```
app.use((req, res, next) => {  
  let start = Date.now()  
  
  res.on("finish", () => {  
    let duration = Date.now() - start  
    console.log(`${req.method} ${req.url} ${res.statusCode}  
    ${duration}ms`)  
  })  
})
```

```
next()  
})
```

Using morgan:

```
npm install morgan
```

```
const morgan = require("morgan")
```

```
app.use(morgan("dev"))
```

Output: GET / 200 15.234 ms - 12

CORS MIDDLEWARE

Enable Cross-Origin Resource Sharing:

```
npm install cors
```

```
const cors = require("cors")
```

```
app.use(cors())
```

With options:

```
app.use(cors({  
  origin: "http://localhost:3000",  
  methods: ["GET", "POST", "PUT", "DELETE"],  
  allowedHeaders: ["Content-Type", "Authorization"]  
}))
```

SECURITY MIDDLEWARE

```
npm install helmet
```

```
const helmet = require("helmet")
```

```
app.use(helmet())
```

Helmet sets security headers:

- Content-Security-Policy
- X-Content-Type-Options
- X-Frame-Options
- And more

COMPRESSION MIDDLEWARE

npm install compression

```
const compression = require("compression")
```

```
app.use(compression())
```

Compresses responses. Reduces bandwidth.

CUSTOM MIDDLEWARE

Request timing:

```
function timing(req, res, next) {  
  req.startTime = Date.now()  
  next()  
}
```

```
app.use(timing)
```

Add request ID:

```
function requestId(req, res, next) {  
  req.id = Date.now().toString(36) +  
    Math.random().toString(36).slice(2)  
  res.setHeader("X-Request-ID", req.id)  
  next()  
}
```

```
app.use(requestId)
```

Validate content type:

```
function requireJson(req, res, next) {  
  if (req.method !== "GET" && !req.is("application/json")) {  
    return res.status(415).json({ error: "Content-Type must be  
    application/json" })  
  }  
  next()  
}
```

```
app.use(requireJson)
```

AUTHENTICATION MIDDLEWARE

```
function authenticate(req, res, next) {  
  let token = req.headers.authorization?.replace("Bearer ", "")  
  
  if (!token) {  
    return res.status(401).json({ error: "No token provided" })  
  }  
  
  try {  
    let user = verifyToken(token)  
    req.user = user  
    next()  
  } catch {  
    res.status(401).json({ error: "Invalid token" })  
  }  
}
```

```
}
```

```
app.use("/api", authenticate)
```

AUTHORIZATION MIDDLEWARE

```
function requireRole(...roles) {  
  return (req, res, next) => {  
    if (!req.user) {  
      return res.status(401).json({ error: "Not authenticated" })  
    }  
  }  
}
```

```
if (!roles.includes(req.user.role)) {  
  return res.status(403).json({ error: "Forbidden" })  
}
```

```
  next()  
}  
}
```

```
app.get("/admin", authenticate, requireRole("admin"), (req, res)  
=> {  
  res.json({ message: "Admin area" })  
})
```

VALIDATION MIDDLEWARE

```
function validate(schema) {  
  return (req, res, next) => {  
    let errors = []  
  
    for (let [field, rules] of Object.entries(schema)) {  
      let value = req.body[field]
```

```
if (rules.required && !value) {  
  errors.push( `${field} is required` )  
}
```

```
if (rules.type && value && typeof value !== rules.type) {  
  errors.push( `${field} must be ${rules.type}` )  
}
```

```
if (rules.minLength && value && value.length < rules.minLength)  
{  
  errors.push( `${field} must be at least ${rules.minLength}  
characters` )  
}  
}
```

```
if (errors.length > 0) {  
  return res.status(400).json({ errors })  
}
```

```
next()  
}  
}
```

```
let userSchema = {  
  name: { required: true, type: "string", minLength: 2 },  
  email: { required: true, type: "string" }  
}
```

```
app.post("/users", validate(userSchema), (req, res) => {  
  res.json(req.body)  
})
```

ERROR-HANDLING MIDDLEWARE

Error middleware has four parameters:

```
function errorHandler(err, req, res, next) {
```

```
console.error(err.stack)
```

```
res.status(err.status ?? 500).json({  
  error: {  
    message: err.message,  
    status: err.status ?? 500  
  }  
})  
}
```

```
app.use(errorHandler)
```

Throw errors in routes:

```
app.get("/users/:id", (req, res, next) => {  
  let user = findUser(req.params.id)
```

```
  if (!user) {  
    let error = new Error("User not found")  
    error.status = 404  
    return next(error)  
  }
```

```
  res.json(user)  
})
```

Or with async:

```
app.get("/users/:id", async (req, res, next) => {  
  try {  
    let user = await findUser(req.params.id)  
    res.json(user)  
  } catch (error) {  
    next(error)  
  }  
})
```

MIDDLEWARE ORDER

Order matters:

```
app.use(express.json())
app.use(morgan("dev"))
app.use(cors())
app.use(helmet())
app.use(compression())

app.use("/api", authenticate)

app.use(routes)

app.use(notFoundHandler)
app.use(errorHandler)
```

Parse body first.

Log requests.

Security and compression.

Authentication.

Routes.

Error handlers last.

ROUTER MIDDLEWARE

Add middleware to router:

```
let router = express.Router()

router.use(authenticate)

router.get("/", (req, res) => {
  res.json([])
})

module.exports = router
```

PRACTICE

Write custom middleware.

Use built-in and third-party middleware.

Implement authentication middleware.

Handle errors with middleware.

Understand middleware order.

WHAT YOU LEARNED

You learned middleware concept and flow.

You learned built-in middleware.

You learned common third-party middleware.

You learned to write custom middleware.

You learned error handling middleware.

You learned middleware ordering.

The next chapter teaches you about request and response objects.

BOOK 14: EXPRESS AND BACKEND

Chapter 5: Request and Response Objects

THE REQUEST OBJECT

`req` is the Express request object.

It extends Node.js IncomingMessage with extra properties.

REQUEST PROPERTIES

req.body - Parsed request body:

```
app.use(express.json())
```

```
app.post("/users", (req, res) => {  
  console.log(req.body.name)  
  console.log(req.body.email)  
})
```

req.params - URL parameters:

```
app.get("/users/:id", (req, res) => {  
  console.log(req.params.id)  
})
```

req.query - Query string parameters:

```
app.get("/search", (req, res) => {  
  console.log(req.query.q)  
  console.log(req.query.page)  
})
```

req.headers - Request headers:

```
app.get("/", (req, res) => {  
  console.log(req.headers.authorization)  
  console.log(req.headers["content-type"])  
  console.log(req.headers.host)  
})
```

req.method - HTTP method:

```
console.log(req.method)
```

req.url - Request URL:

console.log(req.url)

req.path - URL path without query string:

console.log(req.path)

req.originalUrl - Original URL:

console.log(req.originalUrl)

MORE REQUEST PROPERTIES

req.ip - Client IP address:

console.log(req.ip)

req.hostname - Host name:

console.log(req.hostname)

req.protocol - Protocol (http or https):

console.log(req.protocol)

req.secure - Is HTTPS:

console.log(req.secure)

req.xhr - Is AJAX request:

console.log(req.xhr)

req.cookies - Parsed cookies (with cookie-parser):

```
console.log(req.cookies.sessionId)
```

REQUEST METHODS

`req.get(header)` - Get header value:

```
let contentType = req.get("Content-Type")  
let auth = req.get("Authorization")
```

`req.is(type)` - Check content type:

```
if (req.is("json")) {  
  console.log("JSON request")  
}
```

```
if (req.is("application/json")) {  
  console.log("JSON request")  
}
```

`req.accepts(types)` - Check Accept header:

```
let type = req.accepts(["json", "html"])
```

```
if (type === "json") {  
  res.json(data)  
} else if (type === "html") {  
  res.render("view", data)  
}
```

THE RESPONSE OBJECT

`res` is the Express response object.

It extends Node.js `ServerResponse` with extra methods.

RESPONSE METHODS

`res.send(body)` - Send response:

```
res.send("Hello")  
res.send({ name: "Alice" })  
res.send(Buffer.from("Hello"))
```

Automatically sets Content-Type.

`res.json(data)` - Send JSON:

```
res.json({ name: "Alice", age: 25 })
```

Always sets Content-Type to application/json.

`res.status(code)` - Set status code:

```
res.status(404).json({ error: "Not found" })  
res.status(201).json(newUser)
```

Chainable.

`res.sendStatus(code)` - Send status with default message:

```
res.sendStatus(200)  
res.sendStatus(404)  
res.sendStatus(500)
```

MORE RESPONSE METHODS

`res.set(header, value)` - Set header:

```
res.set("X-Custom-Header", "value")
res.set("Content-Type", "text/html")
```

Multiple headers:

```
res.set({
  "X-Custom-Header": "value",
  "Cache-Control": "no-cache"
})
```

res.get(header) - Get response header:

```
let type = res.get("Content-Type")
```

res.type(type) - Set Content-Type:

```
res.type("json")
res.type("html")
res.type("application/pdf")
```

res.redirect(url) - Redirect:

```
res.redirect("/login")
res.redirect(301, "/new-url")
res.redirect("back")
```

FILE RESPONSES

res.sendFile(path) - Send file:

```
const path = require("path")
```

```
app.get("/download", (req, res) => {
  res.sendFile(path.join(__dirname, "files", "document.pdf"))
})
```

With options:

```
res.sendFile(filePath, {  
  headers: { "X-Custom-Header": "value" },  
  dotfiles: "deny"  
})
```

res.download(path, filename) - Force download:

```
app.get("/download", (req, res) => {  
  res.download("./files/report.pdf", "my-report.pdf")  
})
```

res.attachment(filename) - Set Content-Disposition:

```
res.attachment("report.pdf")  
res.send(fileBuffer)
```

COOKIE METHODS

res.cookie(name, value, options) - Set cookie:

```
res.cookie("sessionId", "abc123", {  
  maxAge: 3600000,  
  httpOnly: true,  
  secure: true,  
  sameSite: "strict"  
})
```

Cookie options:

- maxAge - Expiry in milliseconds
- expires - Expiry date
- httpOnly - Not accessible via JavaScript
- secure - HTTPS only
- sameSite - CSRF protection
- path - Cookie path
- domain - Cookie domain

`res.clearCookie(name)` - Clear cookie:

`res.clearCookie("sessionId")`

RESPONSE END

`res.end()` - End response:

`res.end()`

`res.end("Done")`

Usually use `res.send()` or `res.json()` instead.

RESPONSE LOCALS

`res.locals` - Data available to templates:

```
app.use((req, res, next) => {  
  res.locals.user = req.user  
  res.locals.env = process.env.NODE_ENV  
  next()  
})
```

Available in rendered views.

CHAINING RESPONSES

Methods are chainable:

```
res  
  .status(201)
```

```
.set("X-Custom", "value")  
.cookie("token", "abc")  
.json({ success: true })
```

COMMON PATTERNS

Success response:

```
function sendSuccess(res, data, status = 200) {  
  res.status(status).json({  
    success: true,  
    data  
  })  
}
```

```
sendSuccess(res, user, 201)
```

Error response:

```
function sendError(res, message, status = 400) {  
  res.status(status).json({  
    success: false,  
    error: message  
  })  
}
```

```
sendError(res, "Invalid input", 400)
```

Paginated response:

```
function sendPaginated(res, data, page, limit, total) {  
  res.json({  
    data,  
    pagination: {  
      page,  
      limit,  
      total,  
    },  
  })  
}
```

```
pages: Math.ceil(total / limit)
}
})
}
```

CONTENT NEGOTIATION

Respond based on Accept header:

```
app.get("/users/:id", (req, res) => {
  let user = { id: 1, name: "Alice" }

  res.format({
    json: () => res.json(user),
    html: () => res.send(`<h1>${user.name}</h1>`),
    text: () => res.send(user.name),
    default: () => res.status(406).send("Not Acceptable")
  })
})
```

PRACTICE

Access request properties.

Use response methods.

Set headers and cookies.

Send files and redirects.

Implement response helpers.

WHAT YOU LEARNED

You learned request properties and methods.

You learned response methods.

You learned file and redirect responses.

You learned cookie handling.

You learned content negotiation.

You learned response patterns.

The next chapter teaches you about error handling.

BOOK 14: EXPRESS AND BACKEND

Chapter 6: Error Handling

ERRORS IN EXPRESS

Express has built-in error handling.

Errors can occur from:

- Route handlers
- Middleware
- Async operations
- Third-party code

SYNCHRONOUS ERRORS

Thrown errors are caught automatically:

```
app.get("/", (req, res) => {  
  throw new Error("Something broke")  
})
```

Express catches this and calls error handlers.

ASYNCHRONOUS ERRORS

Async errors need explicit handling:

```
app.get("/users/:id", async (req, res, next) => {  
  try {  
    let user = await findUser(req.params.id)  
    res.json(user)  
  } catch (error) {  
    next(error)  
  }  
})
```

Pass error to next() to trigger error handlers.

Without try-catch, unhandled rejections crash the server.

ASYNC WRAPPER

Wrap async handlers:

```
function asyncHandler(fn) {  
  return (req, res, next) => {  
    Promise.resolve(fn(req, res, next)).catch(next)  
  }  
}  
  
app.get("/users/:id", asyncHandler(async (req, res) => {  
  let user = await findUser(req.params.id)  
  res.json(user)  
})))
```

Errors automatically passed to next().

EXPRESS-ASYNC-ERRORS

Or use express-async-errors package:

```
npm install express-async-errors
```

```
require("express-async-errors")  
const express = require("express")
```

```
let app = express()
```

```
app.get("/users/:id", async (req, res) => {  
  let user = await findUser(req.params.id)  
  res.json(user)  
})
```

Async errors handled automatically.

ERROR-HANDLING MIDDLEWARE

Four parameters make it an error handler:

```
app.use((err, req, res, next) => {  
  console.error(err.stack)  
  res.status(500).json({ error: "Something went wrong" })  
})
```

Must be defined after all routes:

```
app.get("/", handler)  
app.post("/users", handler)
```

```
app.use(errorHandler)
```

CUSTOM ERROR CLASS

```
class AppError extends Error {  
  constructor(message, statusCode = 500) {  
    super(message)  
    this.statusCode = statusCode  
    this.status = statusCode >= 400 && statusCode < 500 ? "fail" :  
    "error"  
    this.isOperational = true  
  
    Error.captureStackTrace(this, this.constructor)  
  }  
}
```

```
class NotFoundError extends AppError {  
  constructor(message = "Not found") {  
    super(message, 404)  
  }  
}
```

```
class ValidationError extends AppError {  
  constructor(message = "Validation failed", errors = []) {  
    super(message, 400)  
    this.errors = errors  
  }  
}
```

```
class UnauthorizedError extends AppError {  
  constructor(message = "Unauthorized") {  
    super(message, 401)  
  }  
}
```

USING CUSTOM ERRORS

```
app.get("/users/:id", asyncHandler(async (req, res) => {
  let user = await findUser(req.params.id)

  if (!user) {
    throw new NotFoundError("User not found")
  }

  res.json(user)
}))

app.post("/users", asyncHandler(async (req, res) => {
  let errors = validateUser(req.body)

  if (errors.length > 0) {
    throw new ValidationError("Invalid input", errors)
  }

  let user = await createUser(req.body)
  res.status(201).json(user)
}))
```

COMPREHENSIVE ERROR HANDLER

```
function errorHandler(err, req, res, next) {
  err.statusCode = err.statusCode ?? 500
  err.status = err.status ?? "error"

  if (process.env.NODE_ENV === "development") {
    res.status(err.statusCode).json({
      status: err.status,
      error: err,
      message: err.message,
      stack: err.stack
    })
  } else {
```

```
if (err.isOperational) {
  res.status(err.statusCode).json({
    status: err.status,
    message: err.message
  })
} else {
  console.error("ERROR:", err)

  res.status(500).json({
    status: "error",
    message: "Something went wrong"
  })
}
}
```

HANDLING SPECIFIC ERRORS

```
function errorHandler(err, req, res, next) {
  if (err.name === "ValidationError") {
    return res.status(400).json({
      status: "fail",
      message: err.message,
      errors: err.errors
    })
  }

  if (err.name === "JsonWebTokenError") {
    return res.status(401).json({
      status: "fail",
      message: "Invalid token"
    })
  }

  if (err.name === "TokenExpiredError") {
    return res.status(401).json({
      status: "fail",
```

```
message: "Token expired"
  })
}
```

```
if (err.code === "ENOENT") {
  return res.status(404).json({
    status: "fail",
    message: "File not found"
  })
}
```

```
if (err.code === 11000) {
  return res.status(409).json({
    status: "fail",
    message: "Duplicate entry"
  })
}
```

```
res.status(err.statusCode ?? 500).json({
  status: err.status ?? "error",
  message: err.message ?? "Internal server error"
})
}
```

404 HANDLER

Place before error handler:

```
app.use((req, res, next) => {
  next(new NotFoundError(`Cannot ${req.method} ${req.url}`))
})
```

```
app.use(errorHandler)
```

LOGGING ERRORS

```
function errorHandler(err, req, res, next) {
  let logData = {
    timestamp: new Date().toISOString(),
    method: req.method,
    url: req.url,
    status: err.statusCode ?? 500,
    message: err.message,
    stack: err.stack,
    user: req.user?.id
  }

  if (err.statusCode >= 500) {
    console.error("ERROR:", logData)
  } else {
    console.warn("WARNING:", logData)
  }

  res.status(err.statusCode ?? 500).json({
    status: err.status ?? "error",
    message: err.message
  })
}
```

UNCAUGHT EXCEPTIONS

Handle crashes:

```
process.on("uncaughtException", (err) => {
  console.error("UNCAUGHT EXCEPTION:", err)
  process.exit(1)
})

process.on("unhandledRejection", (reason, promise) => {
  console.error("UNHANDLED REJECTION:", reason)
  process.exit(1)
})
```

```
})
```

Place at start of application.

COMPLETE ERROR SETUP

```
require("express-async-errors")  
const express = require("express")
```

```
class AppError extends Error {  
  constructor(message, statusCode = 500) {  
    super(message)  
    this.statusCode = statusCode  
    this.isOperational = true  
  }  
}
```

```
let app = express()
```

```
app.use(express.json())
```

```
app.get("/users/:id", async (req, res) => {  
  let user = await findUser(req.params.id)
```

```
  if (!user) {  
    throw new AppError("User not found", 404)  
  }
```

```
  res.json(user)  
})
```

```
app.use((req, res, next) => {  
  throw new AppError(`Cannot ${req.method} ${req.url}`, 404)  
})
```

```
app.use((err, req, res, next) => {  
  let statusCode = err.statusCode ?? 500
```

```
res.status(statusCode).json({
  error: {
    message: err.message,
    status: statusCode
  }
})
})

app.listen(3000)
```

PRACTICE

Handle sync and async errors.

Create custom error classes.

Build comprehensive error handler.

Log errors appropriately.

Handle 404 errors.

WHAT YOU LEARNED

You learned sync and async error handling.

You learned error-handling middleware.

You learned custom error classes.

You learned environment-based error responses.

You learned error logging.

You learned 404 handling.

The next chapter teaches you about serving static files.
BOOK 14: EXPRESS AND BACKEND
Chapter 7: Static Files

EXPRESS.STATIC

Serve static files with built-in middleware:

```
app.use(express.static("public"))
```

Files in public/ are accessible:

```
public/  
index.html  
css/  
style.css  
js/  
app.js  
images/  
logo.png
```

Access:

```
http://localhost:3000/index.html  
http://localhost:3000/css/style.css  
http://localhost:3000/images/logo.png
```

MOUNT PATH

Serve under specific path:

```
app.use("/static", express.static("public"))
```

Access:

```
http://localhost:3000/static/css/style.css  
http://localhost:3000/static/images/logo.png
```

ABSOLUTE PATH

Use absolute path for reliability:

```
const path = require("path")

app.use(express.static(path.join(__dirname, "public")))
```

MULTIPLE DIRECTORIES

Serve from multiple directories:

```
app.use(express.static("public"))
app.use(express.static("files"))
app.use(express.static("uploads"))
```

Express checks each directory in order.

OPTIONS

Configure static serving:

```
app.use(express.static("public", {
  dotfiles: "ignore",
  etag: true,
  extensions: ["html", "htm"],
  index: "index.html",
  maxAge: "1d",
  redirect: true
}))
```

dotfiles - Handle dotfiles (ignore, allow, deny)
etag - Enable ETags
extensions - Try extensions for extensionless requests
index - Default file for directories
maxAge - Cache max-age header
redirect - Redirect directories to trailing slash

CACHING

Set cache headers:

```
app.use(express.static("public", {  
  maxAge: "7d"  
}))
```

Different cache for different files:

```
app.use("/images", express.static("public/images", {  
  maxAge: "30d"  
}))
```

```
app.use("/js", express.static("public/js", {  
  maxAge: "1h"  
}))
```

```
app.use(express.static("public", {  
  maxAge: "1d"  
}))
```

CUSTOM HEADERS

Set headers with setHeaders:

```
app.use(express.static("public", {
  setHeaders: (res, path) => {
    if (path.endsWith(".html")) {
      res.set("Cache-Control", "no-cache")
    }

    if (path.endsWith(".js")) {
      res.set("Content-Type", "application/javascript")
    }
  }
}))
```

SERVING INDEX FILES

By default, serves index.html for directories:

```
GET / -> public/index.html
GET /about/ -> public/about/index.html
```

Disable:

```
app.use(express.static("public", {
  index: false
}))
```

Different index file:

```
app.use(express.static("public", {
  index: "default.htm"
}))
```

FALLTHROUGH

Enable/disable 404 handling:

```
app.use(express.static("public", {  
  fallthrough: true  
}))
```

fallthrough: true - Continue to next middleware if not found

fallthrough: false - Return 404 immediately

VIRTUAL PATH

Serve files under different path:

```
app.use("/downloads", express.static("secret-files"))
```

Files in secret-files/ accessible under /downloads/.

SENDING FILES MANUALLY

res.sendFile for single files:

```
const path = require("path")
```

```
app.get("/download/:filename", (req, res) => {  
  let filePath = path.join(__dirname, "files", req.params.filename)
```

```
  res.sendFile(filePath, (err) => {  
    if (err) {  
      res.status(404).send("File not found")  
    }  
  })  
})
```

With options:

```
res.sendFile(filePath, {
  headers: {
    "X-Timestamp": Date.now()
  },
  dotfiles: "deny"
})
```

FORCING DOWNLOADS

res.download forces download:

```
app.get("/download/:filename", (req, res) => {
  let filePath = path.join(__dirname, "files", req.params.filename)
  let downloadName = req.query.name ?? req.params.filename

  res.download(filePath, downloadName, (err) => {
    if (err) {
      res.status(404).send("File not found")
    }
  })
})
```

FILE ATTACHMENTS

res.attachment sets Content-Disposition:

```
app.get("/report", (req, res) => {
  res.attachment("report.pdf")
  res.send(pdfBuffer)
})
```

SPA FALLBACK

For single-page applications:

```
app.use(express.static("public"))
```

```
app.get("*", (req, res) => {  
  res.sendFile(path.join(__dirname, "public", "index.html"))  
})
```

All routes serve index.html. Frontend routing handles the rest.

PROTECTED FILES

Serve files with authentication:

```
function protectedStatic(directory) {  
  let staticMiddleware = express.static(directory)
```

```
  return (req, res, next) => {  
    if (!req.user) {  
      return res.status(401).send("Unauthorized")  
    }  
  }
```

```
  staticMiddleware(req, res, next)  
}  
}
```

```
app.use("/private", protectedStatic("private-files"))
```

FILE UPLOADS DIRECTORY

Serve uploaded files:

```
app.use("/uploads", express.static("uploads"))
```

With security:

```
app.use("/uploads", express.static("uploads", {  
  dotfiles: "deny",  
  setHeaders: (res, path) => {  
    res.set("X-Content-Type-Options", "nosniff")  
  }  
}))
```

COMPRESSION

Compress static files:

```
const compression = require("compression")
```

```
app.use(compression())  
app.use(express.static("public"))
```

Or pre-compress files and serve .gz versions.

COMPLETE STATIC SETUP

```
const path = require("path")  
const compression = require("compression")
```

```
app.use(compression())
```

```
app.use("/assets", express.static(path.join(__dirname, "public/  
assets"), {  
  maxAge: "1y",  
  immutable: true  
}))
```

```
app.use(express.static(path.join(__dirname, "public"), {
  maxAge: "1d",
  etag: true,
  index: "index.html",
  dotfiles: "deny"
}))

app.get("*", (req, res) => {
  res.sendFile(path.join(__dirname, "public", "index.html"))
})
```

PRACTICE

Serve static files with `express.static`.

Configure caching.

Handle file downloads.

Set up SPA fallback.

Protect sensitive files.

WHAT YOU LEARNED

You learned `express.static` middleware.

You learned mount paths and options.

You learned caching configuration.

You learned sending files manually.

You learned SPA fallback pattern.

You learned protected file serving.

The next chapter teaches you about template engines.

BOOK 14: EXPRESS AND BACKEND

Chapter 8: Template Engines

WHAT ARE TEMPLATE ENGINES

Template engines generate HTML from templates and data.

Instead of writing HTML strings in JavaScript:

```
let html = "<h1>Hello, " + name + "!</h1>"
```

You write templates:

```
<h1>Hello, <%= name %>!</h1>
```

Template engines handle:

- Variable interpolation
- Conditionals
- Loops
- Partial and layouts

POPULAR TEMPLATE ENGINES

EJS - Embedded JavaScript, simple syntax

Pug (Jade) - Whitespace-significant, concise

Handlebars - Logic-less, Mustache-based

Nunjucks - Jinja2-like, powerful

We will focus on EJS for its simplicity.

SETTING UP EJS

Install:

```
npm install ejs
```

Configure Express:

```
app.set("view engine", "ejs")
app.set("views", "./views")
```

Create views/index.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
</head>
<body>
  <h1>Welcome, <%= name %>!</h1>
</body>
</html>
```

Render:

```
app.get("/", (req, res) => {
  res.render("index", { title: "Home", name: "Alice" })
})
```

EJS SYNTAX

Output escaped (safe):

```
<%= variable %>
```

Output unescaped (raw HTML):

<%- htmlContent %>

JavaScript code:

```
<% if (condition) { %>
  <p>Condition is true</p>
<% } %>
```

Comment:

<%# This is a comment %>

CONDITIONALS

```
<% if (user) { %>
  <p>Welcome, <%= user.name %>!</p>
<% } else { %>
  <p>Please log in</p>
<% } %>
```

Ternary:

<p class="<%= active ? 'active' : 'inactive' %>">Status</p>

LOOPS

Array iteration:

```
<ul>
<% users.forEach(user => { %>
  <li><%= user.name %> - <%= user.email %></li>
<% }) %>
</ul>
```

For loop:

```
<ul>
<% for (let i = 0; i < items.length; i++) { %>
  <li><%= i + 1 %>. <%= items[i] %> </li>
<% } %>
</ul>
```

PARTIALS

Include other templates:

views/partials/header.ejs:

```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %> </title>
  <link rel="stylesheet" href="/css/style.css">
</head>
<body>
```

views/partials/footer.ejs:

```
<footer>
<p>Copyright 2024 </p>
</footer>
</body>
</html>
```

views/index.ejs:

```
<%- include("partials/header") %>

<main>
<h1>Welcome </h1>
<p>This is the home page. </p>
```

```
</main>
```

```
<%- include("partials/footer") %>
```

PASSING DATA TO PARTIALS

```
<%- include("partials/nav", { currentPage: "home" }) %>
```

In nav.ejs:

```
<nav>
```

```
  <a href="/" class="<%= currentPage === 'home' ? 'active' : "
```

```
  %>">Home</a>  
  <a href="/about" class="<%= currentPage === 'about' ?  
'active' : " %>">About</a>
```

```
</nav>
```

LAYOUTS

Create a layout system:

views/layouts/main.ejs:

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <title> <%= title %> </title>
```

```
  <link rel="stylesheet" href="/css/style.css">
```

```
</head>
```

```
<body>
```

```
  <%- include("../partials/nav") %>
```

```
<main>
```

```
<%- body %>
```

```
</main>
```

```
<%- include("../partials/footer") %>
</body>
</html>
```

Use express-ejs-layouts:

```
npm install express-ejs-layouts
```

```
const expressLayouts = require("express-ejs-layouts")
```

```
app.use(expressLayouts)
app.set("layout", "layouts/main")
```

Views automatically use the layout.

RES.LOCALS

Data available to all views:

```
app.use((req, res, next) => {
  res.locals.user = req.user
  res.locals.year = new Date().getFullYear()
  next()
})
```

In any view:

```
<% if (user) { %>
  <p>Logged in as <%= user.name %> </p>
<% } %>

<p>Copyright <%= year %> </p>
```

HELPER FUNCTIONS

Pass functions to templates:

```
app.use((req, res, next) => {  
  res.locals.formatDate = (date) => {  
    return new Date(date).toLocaleDateString()  
  }  
  
  res.locals.truncate = (str, len) => {  
    return str.length > len ? str.slice(0, len) + "..." : str  
  }  
  
  next()  
})
```

In view:

```
<p>Published: <%= formatDate(post.createdAt) %> </p>  
<p><%= truncate(post.content, 100) %> </p>
```

FORM HANDLING

views/users/new.ejs:

```
<form action = "/users" method = "POST">  
  <div>  
    <label>Name: </label>  
    <input type = "text" name = "name" value = "<%=  
formData?.name ?? " %>">  
    <% if (errors?.name) { %>  
    <span class = "error"> <%= errors.name %> </span>  
    <% } %>  
  </div>  
  
  <div>
```

```

<label> Email: </label>
<input type="email" name="email" value="<%=
formData?.email ?? " %>">
<%= if (errors?.email) { %>
<span class="error"> <%= errors.email %> </span>
<%= } %>
</div>

<button type="submit"> Create </button>
</form>

```

Route:

```

app.get("/users/new", (req, res) => {
  res.render("users/new", { formData: null, errors: null })
})

app.post("/users", (req, res) => {
  let errors = validate(req.body)

  if (Object.keys(errors).length > 0) {
    return res.render("users/new", { formData: req.body, errors })
  }

  createUser(req.body)
  res.redirect("/users")
})

```

FLASH MESSAGES

npm install connect-flash express-session

```

const session = require("express-session")
const flash = require("connect-flash")

```

```

app.use(session({
  secret: "secret",

```

```
resave: false,  
saveUninitialized: false  
}))
```

```
app.use(flash())
```

```
app.use((req, res, next) => {  
  res.locals.success = req.flash("success")  
  res.locals.error = req.flash("error")  
  next()  
})
```

Set flash:

```
app.post("/users", (req, res) => {  
  createUser(req.body)  
  req.flash("success", "User created!")  
  res.redirect("/users")  
})
```

Display in view:

```
<% if (success.length > 0) { %>  
  <div class="alert success"> <%= success %> </div>  
<% } %>
```

```
<% if (error.length > 0) { %>  
  <div class="alert error"> <%= error %> </div>  
<% } %>
```

COMPLETE EXAMPLE

```
const express = require("express")  
const expressLayouts = require("express-ejs-layouts")
```

```
let app = express()
```

```
app.set("view engine", "ejs")
app.set("views", "./views")
app.use(expressLayouts)
app.set("layout", "layouts/main")

app.use(express.urlencoded({ extended: true }))
app.use(express.static("public"))

let users = []

app.get("/", (req, res) => {
  res.render("index", { title: "Home" })
})

app.get("/users", (req, res) => {
  res.render("users/index", { title: "Users", users })
})

app.get("/users/new", (req, res) => {
  res.render("users/new", { title: "New User", errors: {} })
})

app.post("/users", (req, res) => {
  users.push({ id: Date.now(), ...req.body })
  res.redirect("/users")
})

app.listen(3000)
```

PRACTICE

Set up EJS with Express.

Use conditionals and loops.

Create partials and layouts.

Handle form data.

Use flash messages.

WHAT YOU LEARNED

You learned to set up template engines.

You learned EJS syntax.

You learned partials and layouts.

You learned res.locals for shared data.

You learned form handling with views.

You learned flash messages.

The next chapter teaches you about API development.

BOOK 14: EXPRESS AND BACKEND

Chapter 9: API Development

REST API DESIGN

REST APIs follow conventions:

GET /users - List all users

GET /users/:id - Get one user

POST /users - Create user

PUT /users/:id - Replace user

PATCH /users/:id - Update user

DELETE /users/:id - Delete user

Resources are nouns, not verbs.

HTTP methods are verbs.

API STRUCTURE

```
const express = require("express")
let app = express()

app.use(express.json())

let users = []

app.get("/api/users", (req, res) => {
  res.json(users)
})

app.get("/api/users/:id", (req, res) => {
  let user = users.find(u => u.id === req.params.id)

  if (!user) {
    return res.status(404).json({ error: "User not found" })
  }

  res.json(user)
})

app.post("/api/users", (req, res) => {
  let user = {
    id: Date.now().toString(),
    ...req.body,
    createdAt: new Date()
  }

  users.push(user)
  res.status(201).json(user)
})

app.put("/api/users/:id", (req, res) => {
  let index = users.findIndex(u => u.id === req.params.id)

  if (index === -1) {
    return res.status(404).json({ error: "User not found" })
  }
})
```

```

}

users[index] = {
  id: req.params.id,
  ...req.body,
  updatedAt: new Date()
}

res.json(users[index])
})

app.patch("/api/users/:id", (req, res) => {
  let user = users.find(u => u.id === req.params.id)

  if (!user) {
    return res.status(404).json({ error: "User not found" })
  }

  Object.assign(user, req.body, { updatedAt: new Date() })

  res.json(user)
})

app.delete("/api/users/:id", (req, res) => {
  let index = users.findIndex(u => u.id === req.params.id)

  if (index === -1) {
    return res.status(404).json({ error: "User not found" })
  }

  users.splice(index, 1)
  res.status(204).end()
})

```

RESPONSE ENVELOPE

Consistent response format:

```
function success(res, data, status = 200) {  
  res.status(status).json({  
    success: true,  
    data  
  })  
}
```

```
function error(res, message, status = 400) {  
  res.status(status).json({  
    success: false,  
    error: { message, status }  
  })  
}
```

```
app.get("/api/users", (req, res) => {  
  success(res, users)  
})
```

```
app.get("/api/users/:id", (req, res) => {  
  let user = users.find(u => u.id === req.params.id)
```

```
  if (!user) {  
    return error(res, "User not found", 404)  
  }
```

```
  success(res, user)  
})
```

PAGINATION

```
app.get("/api/users", (req, res) => {  
  let page = parseInt(req.query.page) || 1  
  let limit = parseInt(req.query.limit) || 10  
  let offset = (page - 1) * limit
```

```
  let paginatedUsers = users.slice(offset, offset + limit)
```

```
res.json({
  data: paginatedUsers,
  pagination: {
    page,
    limit,
    total: users.length,
    pages: Math.ceil(users.length / limit)
  }
})
})
```

FILTERING

```
app.get("/api/users", (req, res) => {
  let result = [...users]
```

```
  if (req.query.name) {
    result = result.filter(u =>
      u.name.toLowerCase().includes(req.query.name.toLowerCase())
    )
  }
```

```
  if (req.query.email) {
    result = result.filter(u =>
      u.email.toLowerCase().includes(req.query.email.toLowerCase())
    )
  }
```

```
  if (req.query.active !== undefined) {
    let active = req.query.active === "true"
    result = result.filter(u => u.active === active)
  }
```

```
  res.json(result)
})
```

SORTING

```
app.get("/api/users", (req, res) => {  
  let result = [...users]  
  let sort = req.query.sort  
  
  if (sort) {  
    let desc = sort.startsWith("-")  
    let field = desc ? sort.slice(1) : sort  
  
    result.sort((a, b) => {  
      if (a[field] < b[field]) return desc ? 1 : -1  
      if (a[field] > b[field]) return desc ? -1 : 1  
      return 0  
    })  
  }  
  
  res.json(result)  
})
```

INPUT VALIDATION

```
function validateUser(data) {  
  let errors = {}  
  
  if (!data.name || data.name.trim().length < 2) {  
    errors.name = "Name must be at least 2 characters"  
  }  
  
  if (!data.email || !/^[\s@]+@[\s@]+\.[\s@]+  
$/.test(data.email)) {  
    errors.email = "Valid email is required"  
  }  
}
```

```

return {
  isValid: Object.keys(errors).length === 0,
  errors
}
}

app.post("/api/users", (req, res) => {
  let { isValid, errors } = validateUser(req.body)

  if (!isValid) {
    return res.status(400).json({ error: "Validation failed", errors })
  }

  let user = {
    id: Date.now().toString(),
    name: req.body.name.trim(),
    email: req.body.email.toLowerCase(),
    createdAt: new Date()
  }

  users.push(user)
  res.status(201).json(user)
})

```

USING EXPRESS-VALIDATOR

```
npm install express-validator
```

```

const { body, validationResult } = require("express-validator")

let validateUserRules = [
  body("name").trim().isLength({ min: 2 }).withMessage("Name
  required"),
  body("email").isEmail().normalizeEmail().withMessage("Valid
  email required")
]

```

```

function validate(req, res, next) {
  let errors = validationResult(req)

  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() })
  }

  next()
}

app.post("/api/users", validateUserRules, validate, (req, res) => {
  let user = {
    id: Date.now().toString(),
    ...req.body,
    createdAt: new Date()
  }

  users.push(user)
  res.status(201).json(user)
})

```

NESTED RESOURCES

```

app.get("/api/users/:userId/posts", (req, res) => {
  let userPosts = posts.filter(p => p.userId ===
req.params.userId)
  res.json(userPosts)
})

app.post("/api/users/:userId/posts", (req, res) => {
  let post = {
    id: Date.now().toString(),
    userId: req.params.userId,
    ...req.body,
    createdAt: new Date()
  }

```

```
posts.push(post)
res.status(201).json(post)
})
```

VERSIONING

Version in URL:

```
app.use("/api/v1", v1Routes)
app.use("/api/v2", v2Routes)
```

Version in header:

```
app.use("/api", (req, res, next) => {
  let version = req.headers["api-version"] || "1"
  req.apiVersion = version
  next()
})
```

RATE LIMITING

npm install express-rate-limit

```
const rateLimit = require("express-rate-limit")
```

```
let limiter = rateLimit({
  windowMs: 15 * 60 * 1000,
  max: 100,
  message: { error: "Too many requests" }
})
```

```
app.use("/api", limiter)
```

API DOCUMENTATION

Document with comments or OpenAPI:

```
app.get("/api/users", (req, res) => {  
  res.json(users)  
})
```

Create /api-docs endpoint:

```
let apiDocs = {  
  openapi: "3.0.0",  
  info: { title: "My API", version: "1.0.0" },  
  paths: {  
    "/api/users": {  
      get: {  
        summary: "List all users",  
        responses: {  
          200: { description: "Array of users" }  
        }  
      }  
    }  
  }  
}
```

```
app.get("/api-docs", (req, res) => {  
  res.json(apiDocs)  
})
```

COMPLETE API MODULE

```
const express = require("express")  
let router = express.Router()  
  
let items = []
```

```

router.get("/", (req, res) => {
  let { page = 1, limit = 10, sort, ...filters } = req.query

  let result = [...items]

  Object.entries(filters).forEach(([key, value]) => {
    result = result.filter(item => item[key] == value)
  })

  if (sort) {
    let desc = sort.startsWith("-")
    let field = desc ? sort.slice(1) : sort
    result.sort((a, b) => {
      if (a[field] < b[field]) return desc ? 1 : -1
      if (a[field] > b[field]) return desc ? -1 : 1
      return 0
    })
  }

  let start = (page - 1) * limit
  let paginated = result.slice(start, start + parseInt(limit))

  res.json({
    data: paginated,
    pagination: {
      page: parseInt(page),
      limit: parseInt(limit),
      total: result.length
    }
  })
})

module.exports = router

```

PRACTICE

Build REST APIs with all methods.

Implement pagination, filtering, sorting.

Validate input data.

Handle errors consistently.

Document your API.

WHAT YOU LEARNED

You learned REST API conventions.

You learned response formatting.

You learned pagination and filtering.

You learned input validation.

You learned rate limiting.

You learned API documentation.

The next chapter teaches you about authentication.

BOOK 14: EXPRESS AND BACKEND

Chapter 10: Authentication

AUTHENTICATION CONCEPTS

Authentication verifies identity. Who are you?

Authorization determines access. What can you do?

Common approaches:

- Session-based (cookies)
- Token-based (JWT)
- OAuth (third-party)

SESSION-BASED AUTH

npm install express-session

```
const session = require("express-session")
```

```
app.use(session({
  secret: process.env.SESSION_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {
    secure: process.env.NODE_ENV === "production",
    httpOnly: true,
    maxAge: 24 * 60 * 60 * 1000
  }
}))
```

Login:

```
app.post("/login", async (req, res) => {
  let { email, password } = req.body
  let user = await findUserByEmail(email)

  if (!user || !await verifyPassword(password, user.password)) {
    return res.status(401).json({ error: "Invalid credentials" })
  }

  req.session.userId = user.id
  res.json({ message: "Logged in", user: { id: user.id, name:
user.name } })
})
```

Logout:

```
app.post("/logout", (req, res) => {
  req.session.destroy()
  res.json({ message: "Logged out" })
})
```

```
})
```

Check auth:

```
function requireAuth(req, res, next) {  
  if (!req.session.userId) {  
    return res.status(401).json({ error: "Not authenticated" })  
  }  
  next()  
}
```

```
app.get("/profile", requireAuth, async (req, res) => {  
  let user = await findUserById(req.session.userId)  
  res.json(user)  
})
```

JWT AUTHENTICATION

npm install jsonwebtoken bcrypt

```
const jwt = require("jsonwebtoken")  
const bcrypt = require("bcrypt")
```

Password hashing:

```
async function hashPassword(password) {  
  return bcrypt.hash(password, 10)  
}
```

```
async function verifyPassword(password, hash) {  
  return bcrypt.compare(password, hash)  
}
```

Token creation:

```
function createToken(user) {  
  return jwt.sign(  

```

```
{ id: user.id, email: user.email },  
process.env.JWT_SECRET,  
{ expiresIn: "1h" }  
)  
}
```

```
function verifyToken(token) {  
  return jwt.verify(token, process.env.JWT_SECRET)  
}
```

Register:

```
app.post("/register", async (req, res) => {  
  let { name, email, password } = req.body  
  
  let existing = await findUserByEmail(email)  
  
  if (existing) {  
    return res.status(409).json({ error: "Email already registered" })  
  }  
  
  let hashedPassword = await hashPassword(password)  
  
  let user = await createUser({  
    name,  
    email,  
    password: hashedPassword  
  })  
  
  let token = createToken(user)  
  
  res.status(201).json({  
    user: { id: user.id, name: user.name, email: user.email },  
    token  
  })  
})
```

Login:

```
app.post("/login", async (req, res) => {
```

```

let { email, password } = req.body
let user = await findUserByEmail(email)

if (!user || !await verifyPassword(password, user.password)) {
  return res.status(401).json({ error: "Invalid credentials" })
}

let token = createToken(user)

res.json({
  user: { id: user.id, name: user.name, email: user.email },
  token
})
})

```

JWT MIDDLEWARE

```

function authenticate(req, res, next) {
  let header = req.headers.authorization

  if (!header || !header.startsWith("Bearer ")) {
    return res.status(401).json({ error: "No token provided" })
  }

  let token = header.slice(7)

  try {
    let payload = verifyToken(token)
    req.user = payload
    next()
  } catch (error) {
    if (error.name === "TokenExpiredError") {
      return res.status(401).json({ error: "Token expired" })
    }
    res.status(401).json({ error: "Invalid token" })
  }
}

```

```
app.get("/api/profile", authenticate, async (req, res) => {  
  let user = await findUserById(req.user.id)  
  res.json(user)  
})
```

REFRESH TOKENS

Long-lived refresh token, short-lived access token:

```
function createTokens(user) {  
  let accessToken = jwt.sign(  
    { id: user.id },  
    process.env.JWT_SECRET,  
    { expiresIn: "15m" }  
  )  
  
  let refreshToken = jwt.sign(  
    { id: user.id },  
    process.env.REFRESH_SECRET,  
    { expiresIn: "7d" }  
  )  
  
  return { accessToken, refreshToken }  
}  
  
app.post("/login", async (req, res) => {  
  let { email, password } = req.body  
  let user = await findUserByEmail(email)  
  
  if (!user || !await verifyPassword(password, user.password)) {  
    return res.status(401).json({ error: "Invalid credentials" })  
  }  
  
  let { accessToken, refreshToken } = createTokens(user)  
  
  await saveRefreshToken(user.id, refreshToken)
```

```

res.json({ accessToken, refreshToken })
})

app.post("/refresh", async (req, res) => {
  let { refreshToken } = req.body

  try {
    let payload = jwt.verify(refreshToken,
process.env.REFRESH_SECRET)
    let stored = await getRefreshToken(payload.id)

    if (stored !== refreshToken) {
      return res.status(401).json({ error: "Invalid refresh token" })
    }

    let user = await findUserById(payload.id)
    let { accessToken, refreshToken: newRefreshToken } =
createTokens(user)

    await saveRefreshToken(user.id, newRefreshToken)

    res.json({ accessToken, refreshToken: newRefreshToken })
  } catch {
    res.status(401).json({ error: "Invalid refresh token" })
  }
})

```

AUTHORIZATION

Role-based access:

```

function requireRole(...roles) {
  return (req, res, next) => {
    if (!req.user) {
      return res.status(401).json({ error: "Not authenticated" })
    }
  }
}

```

```
if (!roles.includes(req.user.role)) {  
  return res.status(403).json({ error: "Forbidden" })  
}
```

```
next()  
}  
}
```

```
app.get("/admin", authenticate, requireRole("admin"), (req, res)  
=> {  
  res.json({ message: "Admin area" })  
})
```

```
app.get("/users", authenticate, requireRole("admin", "moderator"),  
(req, res) => {  
  res.json(users)  
})
```

RESOURCE OWNERSHIP

```
function requireOwnership(resourceFinder) {  
  return async (req, res, next) => {  
    let resource = await resourceFinder(req.params.id)
```

```
    if (!resource) {  
      return res.status(404).json({ error: "Not found" })  
    }
```

```
    if (resource.userId !== req.user.id && req.user.role !== "admin")  
    {  
      return res.status(403).json({ error: "Forbidden" })  
    }
```

```
    req.resource = resource  
    next()  
  }
```

```

}

app.put("/posts/:id",
  authenticate,
  requireOwnership(findPostById),
  async (req, res) => {
    let updated = await updatePost(req.resource.id, req.body)
    res.json(updated)
  }
)

```

PASSWORD RESET

```

app.post("/forgot-password", async (req, res) => {
  let { email } = req.body
  let user = await findUserByEmail(email)

  if (!user) {
    return res.json({ message: "If email exists, reset link sent" })
  }

  let token = crypto.randomBytes(32).toString("hex")
  await saveResetToken(user.id, token, Date.now() + 3600000)
  await sendResetEmail(email, token)

  res.json({ message: "If email exists, reset link sent" })
})

app.post("/reset-password", async (req, res) => {
  let { token, password } = req.body

  let reset = await findResetToken(token)

  if (!reset || reset.expires < Date.now()) {
    return res.status(400).json({ error: "Invalid or expired token" })
  }
}

```

```
let hashedPassword = await hashPassword(password)
await updateUserPassword(reset.userId, hashedPassword)
await deleteResetToken(token)

res.json({ message: "Password updated" })
})
```

COMPLETE AUTH MODULE

```
const express = require("express")
const jwt = require("jsonwebtoken")
const bcrypt = require("bcrypt")

let router = express.Router()

async function hashPassword(password) {
  return bcrypt.hash(password, 10)
}

async function verifyPassword(password, hash) {
  return bcrypt.compare(password, hash)
}

function createToken(user) {
  return jwt.sign({ id: user.id, role: user.role },
    process.env.JWT_SECRET, { expiresIn: "1h" })
}

function authenticate(req, res, next) {
  let token = req.headers.authorization?.replace("Bearer ", "")

  if (!token) {
    return res.status(401).json({ error: "No token" })
  }

  try {
    req.user = jwt.verify(token, process.env.JWT_SECRET)
```

```

next()
} catch {
res.status(401).json({ error: "Invalid token" })
}
}

router.post("/register", async (req, res) => {
  let { name, email, password } = req.body
  let hash = await hashPassword(password)
  let user = await createUser({ name, email, password: hash })
  let token = createToken(user)
  res.status(201).json({ user: { id: user.id, name, email }, token })
})

router.post("/login", async (req, res) => {
  let { email, password } = req.body
  let user = await findUserByEmail(email)

  if (!user || !await verifyPassword(password, user.password)) {
    return res.status(401).json({ error: "Invalid credentials" })
  }

  let token = createToken(user)
  res.json({ user: { id: user.id, name: user.name, email }, token })
})

router.get("/me", authenticate, async (req, res) => {
  let user = await findUserById(req.user.id)
  res.json({ id: user.id, name: user.name, email: user.email })
})

module.exports = { router, authenticate }

```

PRACTICE

Implement session-based auth.

Implement JWT authentication.

Add refresh tokens.

Implement authorization.

Handle password reset.

WHAT YOU LEARNED

You learned session-based authentication.

You learned JWT authentication.

You learned refresh token pattern.

You learned role-based authorization.

You learned password reset flow.

The next chapter teaches you about project structure.

BOOK 14: EXPRESS AND BACKEND

Chapter 11: Project Structure

WHY STRUCTURE MATTERS

Small apps work with one file.

Large apps need organization:

- Find code quickly
- Separate concerns
- Enable testing
- Support team work
- Scale with growth

BASIC STRUCTURE

```
my-api/  
src/  
app.js  
routes/  
controllers/  
middleware/  
models/  
services/  
utils/  
config/  
tests/  
package.json  
.env
```

ENTRY POINT

src/app.js:

```
const express = require("express")  
require("dotenv").config()  
  
const routes = require("./routes")  
const errorHandler = require("./middleware/errorHandler")  
  
let app = express()  
  
app.use(express.json())  
app.use(express.urlencoded({ extended: true }))  
  
app.use("/api", routes)  
  
app.use(errorHandler)  
  
module.exports = app
```

src/index.js or server.js:

```
const app = require("./app")

let port = process.env.PORT || 3000

app.listen(port, () => {
  console.log(`Server running on port ${port}`)
})
```

Separating app from server allows testing without starting server.

ROUTES

src/routes/index.js:

```
const express = require("express")
let router = express.Router()

router.use("/users", require("./users"))
router.use("/posts", require("./posts"))
router.use("/auth", require("./auth"))
```

```
module.exports = router
```

src/routes/users.js:

```
const express = require("express")
const userController = require("../controllers/userController")
const { authenticate } = require("../middleware/auth")
```

```
let router = express.Router()
```

```
router.get("/", userController.list)
router.get("/:id", userController.get)
router.post("/", authenticate, userController.create)
```

```
router.put("/:id", authenticate, userController.update)
router.delete("/:id", authenticate, userController.remove)

module.exports = router
```

CONTROLLERS

src/controllers/userController.js:

```
const userService = require("../services/userService")
```

```
async function list(req, res, next) {
  try {
    let users = await userService.getAll(req.query)
    res.json(users)
  } catch (error) {
    next(error)
  }
}
```

```
async function get(req, res, next) {
  try {
    let user = await userService.getById(req.params.id)
    res.json(user)
  } catch (error) {
    next(error)
  }
}
```

```
async function create(req, res, next) {
  try {
    let user = await userService.create(req.body)
    res.status(201).json(user)
  } catch (error) {
    next(error)
  }
}
```

```
async function update(req, res, next) {  
  try {  
    let user = await userService.update(req.params.id, req.body)  
    res.json(user)  
  } catch (error) {  
    next(error)  
  }  
}
```

```
async function remove(req, res, next) {  
  try {  
    await userService.remove(req.params.id)  
    res.status(204).end()  
  } catch (error) {  
    next(error)  
  }  
}
```

```
module.exports = { list, get, create, update, remove }
```

SERVICES

src/services/userService.js:

```
const User = require("../models/User")  
const AppError = require("../utils/AppError")
```

```
async function getAll(query) {  
  let { page = 1, limit = 10 } = query  
  return User.findAll({ page, limit })  
}
```

```
async function getById(id) {  
  let user = await User.findById(id)
```

```
  if (!user) {
```

```
throw new AppError("User not found", 404)
}

return user
}

async function create(data) {
  let existing = await User.findByEmail(data.email)

  if (existing) {
    throw new AppError("Email already exists", 409)
  }

  return User.create(data)
}

async function update(id, data) {
  let user = await getById(id)
  return User.update(user.id, data)
}

async function remove(id) {
  let user = await getById(id)
  return User.delete(user.id)
}

module.exports = { getAll, getById, create, update, remove }
```

MODELS

src/models/User.js:

```
let users = []

class User {
  static async findAll({ page, limit }) {
    let start = (page - 1) * limit
```

```
return users.slice(start, start + limit)
}
```

```
static async findById(id) {
return users.find(u => u.id === id)
}
```

```
static async findByEmail(email) {
return users.find(u => u.email === email)
}
```

```
static async create(data) {
let user = {
id: Date.now().toString(),
...data,
createdAt: new Date()
}
```

```
users.push(user)
return user
}
```

```
static async update(id, data) {
let index = users.findIndex(u => u.id === id)
users[index] = { ...users[index], ...data, updatedAt: new Date() }
return users[index]
}
```

```
static async delete(id) {
let index = users.findIndex(u => u.id === id)
users.splice(index, 1)
}
}
```

```
module.exports = User
```

MIDDLEWARE

src/middleware/auth.js:

```
const jwt = require("jsonwebtoken")
const AppError = require("../utils/AppError")

function authenticate(req, res, next) {
  let token = req.headers.authorization?.replace("Bearer ", "")

  if (!token) {
    return next(new AppError("No token provided", 401))
  }

  try {
    req.user = jwt.verify(token, process.env.JWT_SECRET)
    next()
  } catch {
    next(new AppError("Invalid token", 401))
  }
}

function requireRole(...roles) {
  return (req, res, next) => {
    if (!roles.includes(req.user?.role)) {
      return next(new AppError("Forbidden", 403))
    }
    next()
  }
}
```

module.exports = { authenticate, requireRole }

src/middleware/errorHandler.js:

```
function errorHandler(err, req, res, next) {
  let statusCode = err.statusCode || 500

  res.status(statusCode).json({
    error: {
      message: err.message,
```

```
status: statusCode
}
})
}
```

```
module.exports = errorHandler
```

UTILITIES

src/utls/AppError.js:

```
class AppError extends Error {
  constructor(message, statusCode = 500) {
    super(message)
    this.statusCode = statusCode
    this.isOperational = true
  }
}
```

```
module.exports = AppError
```

src/utls/validator.js:

```
function validateRequired(data, fields) {
  let errors = {}

  fields.forEach(field => {
    if (!data[field]) {
      errors[field] = `${field} is required`
    }
  })

  return {
    isValid: Object.keys(errors).length === 0,
    errors
  }
}
```

```
module.exports = { validateRequired }
```

CONFIG

config/index.js:

```
module.exports = {  
  port: process.env.PORT || 3000,  
  nodeEnv: process.env.NODE_ENV || "development",  
  jwt: {  
    secret: process.env.JWT_SECRET,  
    expiresIn: process.env.JWT_EXPIRES || "1h"  
  },  
  database: {  
    url: process.env.DATABASE_URL  
  }  
}
```

FEATURE-BASED STRUCTURE

Alternative organization by feature:

```
src/  
  features/  
    users/  
      userRoutes.js  
      userController.js  
      userService.js  
      userModel.js  
      userValidator.js  
    posts/  
      postRoutes.js  
      postController.js
```

```
postService.js
postModel.js
auth/
authRoutes.js
authController.js
authService.js
shared/
middleware/
utils/
app.js
```

DEPENDENCY INJECTION

For testability:

```
class UserService {
  constructor(userRepository) {
    this.userRepository = userRepository
  }

  async getById(id) {
    return this.userRepository.findById(id)
  }
}
```

```
let userService = new UserService(new UserRepository())
```

Or use a container:

```
const container = {
  userRepository: new UserRepository(),
  userService: null
}
```

```
container.userService = new UserService(container.userRepository)
```

PRACTICE

Organize routes and controllers.

Separate business logic into services.

Create reusable middleware.

Use configuration files.

Consider feature-based structure.

WHAT YOU LEARNED

You learned standard folder structure.

You learned to separate routes, controllers, services.

You learned middleware organization.

You learned utility and config patterns.

You learned feature-based alternative.

The next chapter brings everything together with practice programs.

BOOK 14: EXPRESS AND BACKEND

Chapter 12: Practice Express Programs

COMPLETE REST API

Full API with authentication and CRUD:

```
const express = require("express")
const jwt = require("jsonwebtoken")
```

```
const bcrypt = require("bcrypt")
require("dotenv").config()
```

```
let app = express()
```

```
app.use(express.json())
```

```
let users = []
```

```
let posts = []
```

```
class AppError extends Error {
  constructor(message, status = 500) {
    super(message)
    this.status = status
  }
}
```

```
function asyncHandler(fn) {
  return (req, res, next) => Promise.resolve(fn(req, res,
next)).catch(next)
}
```

```
function authenticate(req, res, next) {
  let token = req.headers.authorization?.replace("Bearer ", "")
```

```
  if (!token) {
    return next(new AppError("No token", 401))
  }
```

```
  try {
    req.user = jwt.verify(token, process.env.JWT_SECRET)
    next()
  } catch {
    next(new AppError("Invalid token", 401))
  }
}
```

```
function createToken(user) {
  return jwt.sign({ id: user.id, email: user.email },
process.env.JWT_SECRET, { expiresIn: "1h" })
}
```

```
}
```

```
app.post("/api/register", asyncHandler(async (req, res) => {  
  let { name, email, password } = req.body
```

```
  if (users.find(u => u.email === email)) {  
    throw new AppError("Email exists", 409)  
  }
```

```
  let user = {  
    id: Date.now().toString(),  
    name,  
    email,  
    password: await bcrypt.hash(password, 10),  
    createdAt: new Date()  
  }
```

```
  users.push(user)
```

```
  let token = createToken(user)
```

```
  res.status(201).json({  
    user: { id: user.id, name, email },  
    token  
  })  
}))
```

```
app.post("/api/login", asyncHandler(async (req, res) => {  
  let { email, password } = req.body  
  let user = users.find(u => u.email === email)
```

```
  if (!user || !await bcrypt.compare(password, user.password)) {  
    throw new AppError("Invalid credentials", 401)  
  }
```

```
  let token = createToken(user)
```

```
  res.json({  
    user: { id: user.id, name: user.name, email },  
    token
```

```
})  
}))
```

```
app.get("/api/me", authenticate, (req, res) => {  
  let user = users.find(u => u.id === req.user.id)  
  res.json({ id: user.id, name: user.name, email: user.email })  
})
```

```
app.get("/api/posts", (req, res) => {  
  let { page = 1, limit = 10, author } = req.query  
  let result = [...posts]
```

```
  if (author) {  
    result = result.filter(p => p.authorId === author)  
  }
```

```
  let start = (page - 1) * limit  
  let paginated = result.slice(start, start + parseInt(limit))
```

```
  res.json({  
    data: paginated.map(p => ({  
      ...p,  
      author: users.find(u => u.id === p.authorId)  
    })),  
    pagination: {  
      page: parseInt(page),  
      limit: parseInt(limit),  
      total: result.length  
    }  
  })  
})
```

```
app.get("/api/posts/:id", (req, res, next) => {  
  let post = posts.find(p => p.id === req.params.id)
```

```
  if (!post) {  
    return next(new AppError("Post not found", 404))  
  }
```

```
  res.json({
```

```
...post,  
author: users.find(u => u.id === post.authorId)  
})  
})
```

```
app.post("/api/posts", authenticate, (req, res) => {  
  let post = {  
    id: Date.now().toString(),  
    ...req.body,  
    authorId: req.user.id,  
    createdAt: new Date()  
  }  

```

```
  posts.push(post)  
  res.status(201).json(post)  
})
```

```
app.put("/api/posts/:id", authenticate, (req, res, next) => {  
  let post = posts.find(p => p.id === req.params.id)
```

```
  if (!post) {  
    return next(new AppError("Post not found", 404))  
  }
```

```
  if (post.authorId !== req.user.id) {  
    return next(new AppError("Forbidden", 403))  
  }
```

```
  Object.assign(post, req.body, { updatedAt: new Date() })  
  res.json(post)  
})
```

```
app.delete("/api/posts/:id", authenticate, (req, res, next) => {  
  let index = posts.findIndex(p => p.id === req.params.id)
```

```
  if (index === -1) {  
    return next(new AppError("Post not found", 404))  
  }
```

```
  if (posts[index].authorId !== req.user.id) {
```

```
return next(new AppError("Forbidden", 403))
}
```

```
posts.splice(index, 1)
res.status(204).end()
})
```

```
app.use((req, res) => {
  res.status(404).json({ error: "Not found" })
})
```

```
app.use((err, req, res, next) => {
  res.status(err.status || 500).json({ error: err.message })
})
```

```
app.listen(3000, () => console.log("Server running on port 3000"))
```

TODO API WITH FILTERING

```
const express = require("express")
let app = express()
```

```
app.use(express.json())
```

```
let todos = []
```

```
app.get("/api/todos", (req, res) => {
  let result = [...todos]
```

```
  if (req.query.completed !== undefined) {
    let completed = req.query.completed === "true"
    result = result.filter(t => t.completed === completed)
  }
```

```
  if (req.query.priority) {
    result = result.filter(t => t.priority === req.query.priority)
  }
```

```
if (req.query.search) {  
  let search = req.query.search.toLowerCase()  
  result = result.filter(t => t.title.toLowerCase().includes(search))  
}
```

```
if (req.query.sort) {  
  let desc = req.query.sort.startsWith("-")  
  let field = desc ? req.query.sort.slice(1) : req.query.sort
```

```
  result.sort((a, b) => {  
    if (a[field] < b[field]) return desc ? 1 : -1  
    if (a[field] > b[field]) return desc ? -1 : 1  
    return 0  
  })  
}
```

```
res.json({  
  data: result,  
  stats: {  
    total: todos.length,  
    completed: todos.filter(t => t.completed).length,  
    active: todos.filter(t => !t.completed).length  
  }  
})  
)
```

```
app.post("/api/todos", (req, res) => {  
  let todo = {  
    id: Date.now().toString(),  
    title: req.body.title,  
    completed: false,  
    priority: req.body.priority || "medium",  
    createdAt: new Date()  
  }  
}
```

```
  todos.push(todo)  
  res.status(201).json(todo)  
})
```

```
app.patch("/api/todos/:id", (req, res) => {  
  let todo = todos.find(t => t.id === req.params.id)  
  
  if (!todo) {  
    return res.status(404).json({ error: "Todo not found" })  
  }  
  
  Object.assign(todo, req.body, { updatedAt: new Date() })  
  res.json(todo)  
})
```

```
app.patch("/api/todos/:id/toggle", (req, res) => {  
  let todo = todos.find(t => t.id === req.params.id)  
  
  if (!todo) {  
    return res.status(404).json({ error: "Todo not found" })  
  }
```

```
  todo.completed = !todo.completed  
  todo.updatedAt = new Date()
```

```
  res.json(todo)  
})
```

```
app.delete("/api/todos/:id", (req, res) => {  
  let index = todos.findIndex(t => t.id === req.params.id)
```

```
  if (index === -1) {  
    return res.status(404).json({ error: "Todo not found" })  
  }
```

```
  todos.splice(index, 1)  
  res.status(204).end()  
})
```

```
app.delete("/api/todos", (req, res) => {  
  if (req.query.completed === "true") {  
    todos = todos.filter(t => !t.completed)  
  } else {  
    todos = []  
  }
```

```
}

res.status(204).end()
})

app.listen(3000)
```

FILE UPLOAD API

```
const express = require("express")
const multer = require("multer")
const path = require("path")
const fs = require("fs").promises

let app = express()

let storage = multer.diskStorage({
  destination: "uploads/",
  filename: (req, file, cb) => {
    let uniqueName = Date.now() + "-" +
    Math.round(Math.random() * 1E9)
    cb(null, uniqueName + path.extname(file.originalname))
  }
})

let upload = multer({
  storage,
  limits: { fileSize: 5 * 1024 * 1024 },
  fileFilter: (req, file, cb) => {
    let allowed = [".jpg", ".jpeg", ".png", ".gif", ".pdf"]
    let ext = path.extname(file.originalname).toLowerCase()

    if (allowed.includes(ext)) {
      cb(null, true)
    } else {
      cb(new Error("Invalid file type"))
    }
  }
})
```

```
}  
})
```

```
let files = []
```

```
app.use("/uploads", express.static("uploads"))
```

```
app.get("/api/files", (req, res) => {  
  res.json(files)  
})
```

```
app.post("/api/files", upload.single("file"), (req, res) => {  
  if (!req.file) {  
    return res.status(400).json({ error: "No file uploaded" })  
  }  
}
```

```
  let fileInfo = {  
    id: Date.now().toString(),  
    originalName: req.file.originalname,  
    filename: req.file.filename,  
    mimetype: req.file.mimetype,  
    size: req.file.size,  
    url: `/uploads/${req.file.filename}`,  
    uploadedAt: new Date()  
  }
```

```
  files.push(fileInfo)  
  res.status(201).json(fileInfo)  
})
```

```
app.post("/api/files/multiple", upload.array("files", 10), (req, res)  
=> {  
  let uploaded = req.files.map(file => ({  
    id: Date.now().toString() + Math.random(),  
    originalName: file.originalname,  
    filename: file.filename,  
    url: `/uploads/${file.filename}`,  
    uploadedAt: new Date()  
  })))
```

```

files.push(...uploaded)
res.status(201).json(uploaded)
})

app.delete("/api/files/:id", async (req, res) => {
  let index = files.findIndex(f => f.id === req.params.id)

  if (index === -1) {
    return res.status(404).json({ error: "File not found" })
  }

  let file = files[index]

  try {
    await fs.unlink(path.join("uploads", file.filename))
  } catch {}

  files.splice(index, 1)
  res.status(204).end()
})

app.use((err, req, res, next) => {
  if (err instanceof multer.MulterError) {
    return res.status(400).json({ error: err.message })
  }
  res.status(500).json({ error: err.message })
})

app.listen(3000)

```

WHAT YOU BUILT

You built three complete Express applications:

- REST API with authentication, CRUD, pagination
- Todo API with filtering and sorting
- File upload API with validation

Each program demonstrated:

Express routing and middleware

JWT authentication

Error handling

Input validation

File handling

WHAT COMES NEXT

You now understand Express and backend development:

- Express basics (Book 14, Chapters 1-2)
- Routing (Book 14, Chapter 3)
- Middleware (Book 14, Chapter 4)
- Request/Response (Book 14, Chapter 5)
- Error handling (Book 14, Chapter 6)
- Static files (Book 14, Chapter 7)
- Templates (Book 14, Chapter 8)
- API development (Book 14, Chapter 9)
- Authentication (Book 14, Chapter 10)
- Project structure (Book 14, Chapter 11)

Book 15 teaches you about databases. You will:

- Understand database concepts
- Use SQL databases
- Use MongoDB
- Work with ORMs
- Design schemas

You finished Book 14. You build APIs. Now you persist data.